05 | 性能方案：你的方案是否还停留在形式上？

2021-03-31 高楼

高楼的性能工程实战课 进入课程 >

讲述：高楼
时长 05:24 大小 4.96M

你好，我是高楼。

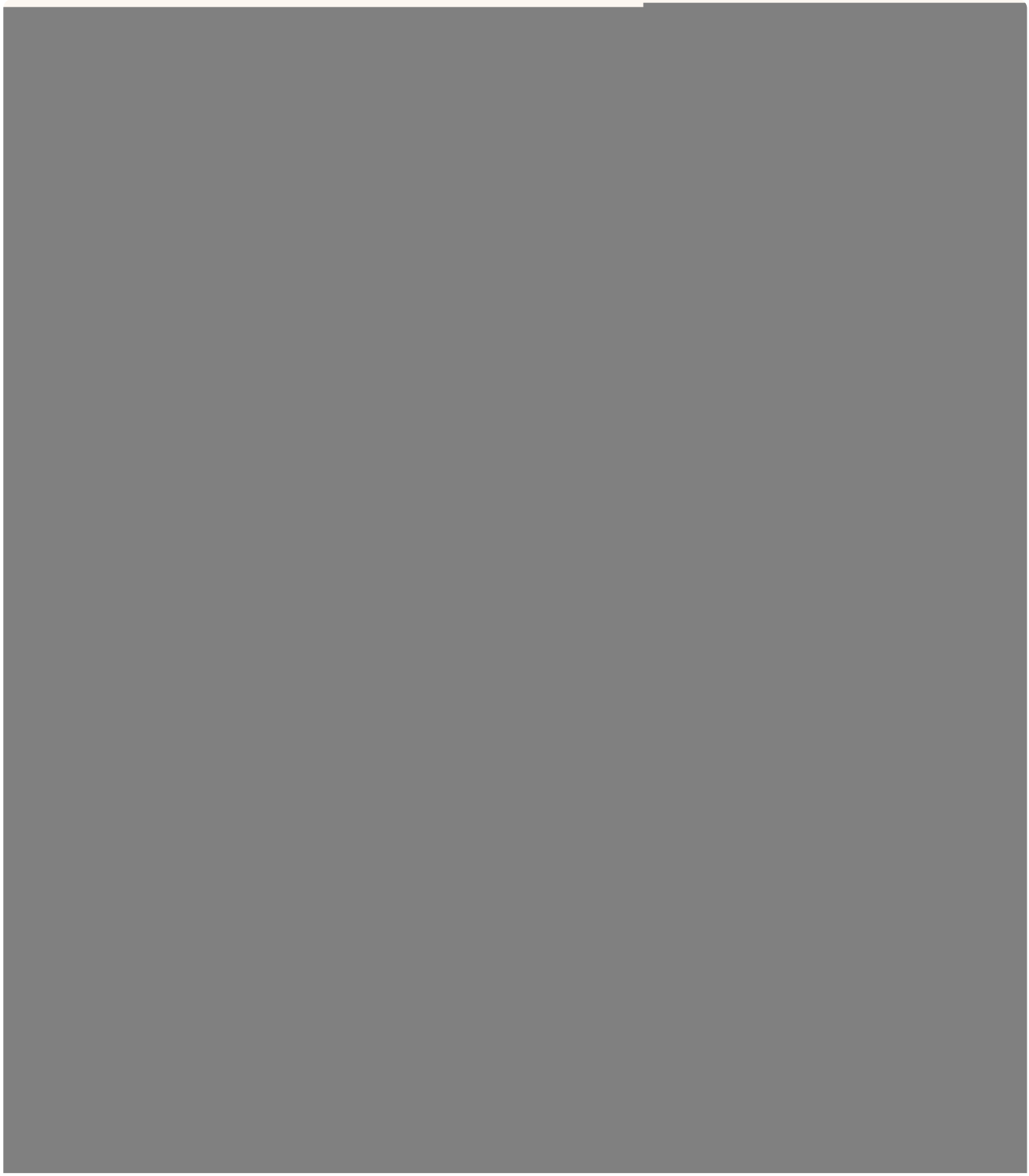
性能方案在性能项目中是非常重要的文档之一，它指导着整个项目的执行过程，同时也约束着项目的边界，定义相关人员的职能。但令人痛心的是，如今它变得“微不足道”。

在很多常见的性能项目中，性能方案就是一个文档，并且是一个静态的文档。里面写的东西是什么，项目后续会不会按这个内容去做，基本上没有人关心。它就成了一个形式，只有在评审方案的时候才会被拿出来看看。甚至在一些第三方测试项目中，我看到有些甲方连方案的内容都不看，直接问有没有。如果有就过去了。你看，一个必需的交付物却无人关心。

在我的性能工程理念中，性能方案是一个重量级的文档。在性能项目中，它被叫成是“性能测试方案”。在我这里，我要把“测试”二字拿掉。为什么要拿掉？因为这取决于我在前面课程中提到的性能工程理念，我希望把整个项目的过程都描述在方案中。

我讲的性能方案和那些常见的性能方案究竟有什么区别呢？我们不妨先来看看，后者普遍都是什么样的。

这些目录相信你并不陌生，我们经常能看到有这样目录的性能测试方案。



这里我就不一一列举了，再看更多的目录，其实也是类似的。这样的目录大纲，在我看来分为这么几个部分。

常规项目信息：比如说测试背景、测试范围、测试准则、测试环境、实施准备、组织结构、项目风险、里程碑。

性能实施信息：比如说测试模型、测试策略、监控策略。

项目输出：比如说测试脚本、测试用例 / 测试场景、监控采集数据，测试报告、调优报告。

从性能测试方案的角度来说，这些内容似乎够了。但是，如果抛弃掉“测试”这个视角，从一个完整的性能项目的角度来看，这些内容其实还不够。

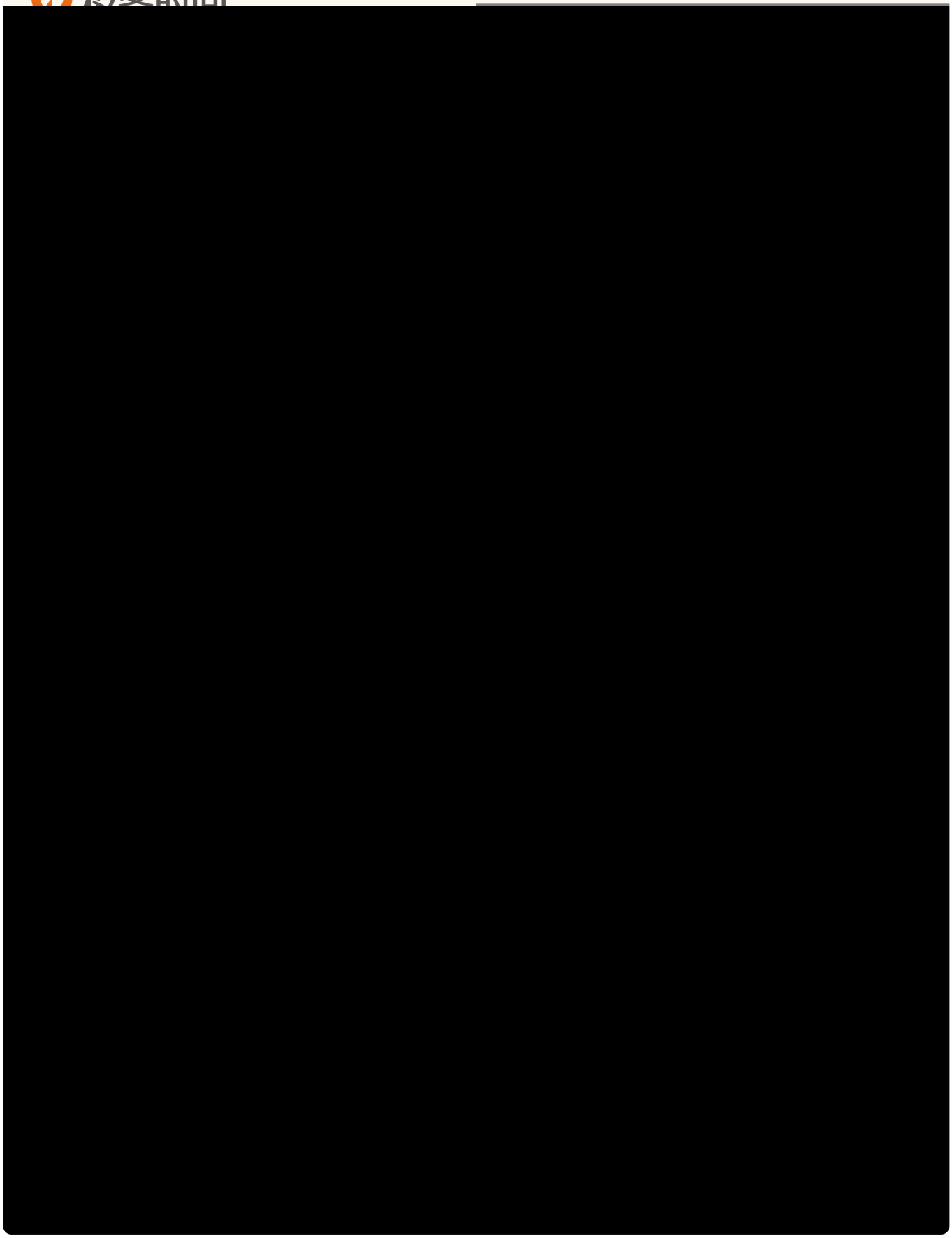
以前经常有人问我要一个性能项目方案模板，我一直不太理解，就这么一个目录，为什么还非得要呢？自己一个字一个字也照样写得出来吧。后来我慢慢理解了，他们要的其实不是大纲目录，而是一个完整的性能方案内容。

不过，我们知道，项目实施的性能方案基本上都不太可能直接发出来，即便做了脱敏，一些内容也可以看出是属于某些企业的。所以，出于职场的素养，这些内容不得不放在硬盘里，直到过时，直到烂掉。这也就是为什么我们在网上看不到非常完整的性能方案。

可是，尽管网上的方案不完整，在性能市场上，我们还是看到有太多的性能方案是抄来抄去的，总体的结构大同小异。这也就导致了在性能项目中，大量的方案都只有形式上的意义。

因为我们这个课程需要基于一个完整的项目来编写，所以，我把这个项目整体的方案写在这里。你将看到，我认为的真正完整并且有意义的性能方案是什么样子，希望能给你一些启发。

由于性能方案的内容比较多，并且相对琐碎，我给你整理了一张性能方案的目录表格，你可以对应这张表格，来学习具体的内容。



性能项目实施方案

背景

项目背景

我们刚才提到，这个课程需要搭建一个比较完整的性能项目。但由于各企业的商业软件有限制，我们只能选择一个开源的项目，并且这个项目最好可以覆盖常见的技术栈，以便能给你提供更多可借鉴的内容。

基于上述原因，我们搭建了一套电商项目。对此，我要说明两点：第一，这个项目是较为完整的；第二，当前电商的系统比较典型，并且这个项目完全开源，便于我们改造。

不过，也因为这是一个开源的项目，功能和性能都不知道会有什么样的问题，我们只有在性能实施的过程中一步步去发掘，所以这是一个非常符合我们当前目标的项目。

性能目标

1. 根据经典的电商下单流程，测试当前系统的单接口最大容量。
2. 根据业务比例设计容量场景，充分利用当前资源，找到当前系统的性能瓶颈，并优化，以达到系统的最佳运行状态。
3. 根据稳定性场景，判断当前系统可支持的系统最大累加容量。
4. 根据异常场景，判断当前系统中的异常对性能产生的影响。

在每一个性能项目中，性能目标都会影响项目的整个过程。因此，对目标的把握将决定一个性能项目的走向。

记得我在之前的一个项目中，客户方要求做到支持 1000 万人在线，项目不算小，开发团队有 300 人左右。到那里后，我一看只有两个性能测试人员，而且其中一个还是刚毕业，还处于打野练级的状态。于是，我就过去找他们科技部的老大说，这个项目我做不了。因为根据这个目标和这样的人员配置，我清楚这个坑根本不是我能填得上的，所以得赶紧认怂。

后来，那个科技部的老大问，需要什么样的资源才能做下去呢？于是我提了几个必需的条件，直到这些条件都满足了，我才敢接这个项目。

我举这个例子是想让你明白，性能目标在上下级眼中根本是不一样的，而我这样的处理，是希望把性能目标在上下级的脑袋中变得一致。这一点很重要。

测试范围

需要测试的特性

电商主流程，如下所示：

不需要测试的特性

批量业务。

准则

启动准则

1. 确定系统逻辑架构和部署架构和生产一致。
2. 确定基础数据和生产一致或按模型缩放。
3. 确定业务模型可以模拟生产真实业务。
4. 环境准备完毕，包括：
   4.1. 功能验证通过。
   4.2. 各组件基础参数梳理并配置正确。
   4.3. 压力机到位，并部署完毕。
   4.4. 网络配置正确，连接通畅，可以满足压力测试需求。
5. 测试计划、方案评审完毕。
6. 架构组、运维组、开发组、测试组及相关专家人员到位。

结束准则

1. 达到项目要求的性能需求指标。
2. 关键性能瓶颈已解决。
3. 完成性能测试报告和性能调优报告。

暂停 / 再启动准则

1. 暂停准则

系统环境变化：举例：系统主机硬件损坏、网络传输时间超长、压力发生器出现损坏、系统主机因别的原因需升级暂停等。

测试环境受到干扰，比如服务器被临时征用，或服务器的其他使用会对测试结果造成干扰。

需要调整测试环境资源，如操作系统、数据库参数等。

该测试机型无法达到规划指标要求。

出现测试风险中列出的问题。

2. 再启动准则

测试中发现问题得以解决。

测试环境恢复正常。

测试风险中出现的问题已解决。

环境调整完毕。

业务模型和性能指标

业务模型 / 测试模型

请你注意，这个模型并不是随便填写的，而是直接从生产环境中取得的业务比例。关于如何从生产中取出这样的业务比例，有很多种手段。这个并不复杂，通过统计日志就可以做到。

不过，在有些企业中，生产数据都在运维手里，性能团队怎么也得不到，因为没有权限，就连做业务模型的数据都没有。如果是这样的话，那性能项目是可以直接终止的，因为做了也没有多大的意义，最多也就是找那些瞎吹牛的架构师和乱写代码的开发人员，犯的一些错而已。

业务指标 / 性能指标

序号	操作	接口	目标TPS	TPS标准方差	响应时间	响应时间标准方差
1	打开首页	/home/content	330	5%	200ms	5%
2	用户登陆	/sso/login (这个接口还会调用/oauth/token、 /sso/loadByUsername)	150	5%	200ms	5%
3	用户信息查询	/member/address/list	150	5%	200ms	5%
4	商品查询（关键字）	/mall-search/esProduct/search	1620	5%	200ms	5%
5	商品加入购物车	/cart/add	150	5%	200ms	5%
6	查询购物车信息	/cart/list	90	5%	200ms	5%
7	购物车信息确定订单	/order/generateConfirmOrder	90	5%	200ms	5%
8	生成订单信息	/order/generateOrder	90	5%	200ms	5%
9	支付前查询订单列表	/order/list	90	5%	200ms	5%
10	支付订单信息	/order/paySuccess	90	5%	200ms	5%
11	支付后查询订单详情	/order/detail/${orderId}	150	5%	200ms	5%

在不清楚项目目标 TPS 的情况下，我们暂定目标 TPS 为 1000。为什么暂定 1000 呢？因为根据经验来说，在这样的硬件环境下，定为 1000 并不算高，除非是没有合理的软件架构。

系统架构图

系统技术栈

系统技术栈是让我们知道整个架构中用了哪些技术组件。而这些技术组件中有哪些常见的性能瓶颈点，有哪些性能参数，我们都可以在查看技术栈时得到一些相关信息。而在后续的工作中，我们也要整理出相应的关键性能参数配置。

下面这张表格，就是我们在后续课程的案例分析中，会用到的技术栈。我在搭建这个系统时，考虑的是尽量覆盖当前技术市场中的主流技术组件。

系统逻辑架构图

画系统的逻辑架构图是为了后续性能分析的时候，脑子里能有一个业务路径。我们在做性能分析时，要做响应时间的拆分，而只有了解了逻辑架构图才可以知道从哪里拆到哪里。

系统部署架构图

画部署架构图是为了让我们知道有多少节点、多少机器。在执行容量场景时，你的脑子里要有一个概念，就是这样的部署架构最大应该可以支持多少的容量上限。

此外，对一些无理的性能需求，你看了部署架构之后，其实就可以拒绝。比如说前段时间有个人跟我说，他们有一个 CRM 系统，在做性能的时候，说要达到 1 万的并发用户。而实际上，那个系统就算是上线了，总用户数可能都不到 1 万。

性能实施前提条件

硬件环境

通过对整体硬件资源的整理，我们可以根据经验知道容量大概能支持多少的业务量级，而不至于随便定无理的指标。比如说，当看到下面表格中这样的硬件配置，我想没有人会把指标定为 10000TPS。因为即使是对于最基础的接口层来说，这样的硬件也支持不了这么大的 TPS。

序号	主机	用途	配置	备注
1	物理服务器1	KVM服务器	16C-32G-280G	root@dell-server-2 ~# free -h total used free shared buff/cache available Mem: 31G 29G 222M 8.4M 1.0G 711M Swap: 63G 13G 50G [root@dell-server-2 ~]#
2	物理服务器2	KVM服务器	16C-32G-280G	root@dell-server-3 ~# free -h total used free shared buff/cache available Mem: 31G 30G 190M 4.2M 437M 163M Swap: 63G 26G 37G [root@dell-server-3 ~]#
				root@hp-server ~# free -h total used free shared buff/cache available

我们可以看到，当前服务器总共使用在应用中的资源是：64C 的 CPU 资源，128G 的内存资源。NFS 服务器不用在应用中，故不计算在内。因为单台机器的硬件资源相对较多，所以在后续的工作中，我们可以将这些物理机化为虚拟机使用，以方便应用的管理。

在成本上，所有物理机加在一起大概 8 万左右的费用，这其中还包括交换机、机柜、网线等各类杂七杂八的费用。

我之所以会对硬件的成本进行一个说明，主要是因为在当前的性能行业中，很少有性能工程师去做成本的计算。我们说性能项目的目标是让线上的系统运行得更好，与此同时，我们也要知道使用了多少成本在运行业务系统。

在当前的性能行业中，有大量的线上主机处于高成本低使用率的状态当中，这是极大的资源浪费和成本消耗。我经常在性能项目中，看到一台 256C512G 的硬件服务器里，只运行了一个 4G JVM 的 Tomcat，性能工程的价值完全没有在这样的项目中应用起来。

因此，我时常会痛心疾首地感慨性能行业的不景气：

企业中没有意识到性能的价值，觉得摆个高配置的硬件服务器，业务系统的性能就能好起来。其实完全不是这样。

性能市场从业人员完全没有把性能的价值，透明化地体现出来。并且很多性能人员自身的技术能力不足，这也让一个企业完全看不到性能本该有的价值。

鉴于此，**作为性能从业人员，我们必须要了解硬件配置和整体业务容量之间的关系。**

工具准备

测试工具

在测试过程中，我们将使用 JMeter 的 backend Listener 把数据直接发到 InfluxDB 中，然后再由 Grafana 来展现。我们不使用 JMeter 的分布式执行功能或本地收集数据的功能，因为这样会消耗本地的 IO。

然而，现在还是有很多性能人员，仍然在项目中频繁地使用那些性能工具的低性能操作手段，同时还在不断抱怨性能这么差。对于这种现状，我希望你可以明白一点：我们要理智地使用工具，不要觉得一个性能测试工具拿起来就可以用。

监控工具

根据 RESAR 性能工程中的全局 - 定向的监控思路，我们在选择第一层监控工具时，要采集全量的全局计数器，采集的计数器包括各个层级，这里请参考前面的架构图。

但是，请你注意，在全局监控中，我们要尽量避免使用定向的监控手段，比如说 java 应用中的方法级监控、数据库中的 SQL 监控等。因为在项目开始之初，我们不能确定到底在哪个层面会出现问题，所以不适合使用定向监控思路。

那全局监控怎么来做才最合理呢？这里我们可以参考线上运维的监控手段。注意，我们在性能监控过程中，尽量不要自己臆想，随意搭建监控工具。

有时我们可能为了能监控得更多，会在测试环境中用很多监控手段。但实际上，线上运维时并不用那些手段，这就导致了监控对资源的消耗大于生产环境的资源消耗，我们也就得不到正常有效的结果。

前面我们提到在选择第一层监控工具时，需要采集全量的全局计数器。在我们采集好全局的计数器后，还需要分析并发现性能问题，然后再通过查找证据链的思路，来找性能瓶颈的根本原因。

数据准备

基础数据

在性能工程中，我们一直强调基础数据要满足两个特性：

1. **满足生产环境的真实数据分布**：要想做到这一点，最合理的方式是脱敏生产数据。如果你要自己造数据的话，也一定要先分析业务逻辑。在我们这个系统中，我造了 243 万条用户数据和 250 万条地址数据。
2. **参数化数据一定要使用基础数据来覆盖真实用户**：一直以来，很多人都在使用少量数据做大压力，这种逻辑是完全不对的。在性能脚本中一定要用基础数据来做参数化，而用多少数据取决于性能场景的设计。

性能设计

场景执行策略

场景递增策略

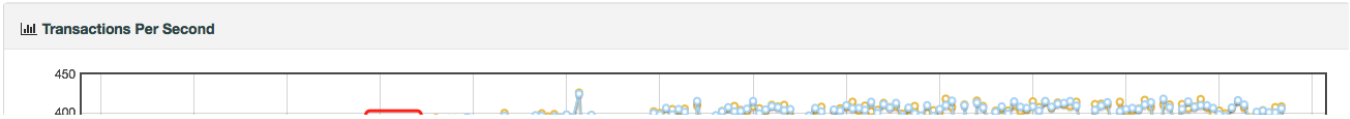
对于性能场景，我一直在强调一个观点，那就是性能的场景必须满足两个条件：

连续

递增

所以在这次的执行过程中，我也会把这两点应用到下面的业务场景中。

你也许会问，如果不连续递增的话会有什么问题呢？比如说下面这样的图：



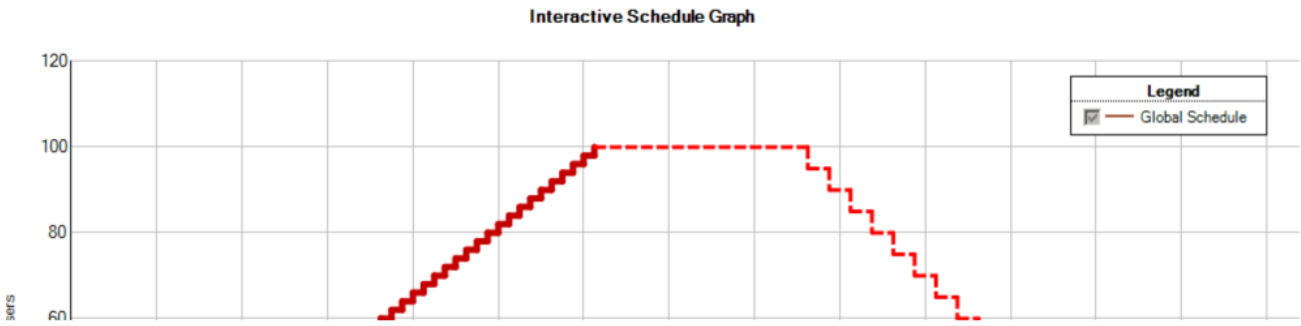
在图中画红框的地方，其实就是递增带来的性能问题表现。因为在递增过程中，被测系统的资源要动态分配。系统会不会在这个时候抖动，我们完全可以从这样的图中看出来，而这样的场景才是真实的线上场景。

如果不连续递增，就不会有图中红框这样的部分。当然了，要是不连续递增，也就不能模拟出线上的真实场景。

高老师画重点！敲黑板了！**要模拟生产场景，连续递增一定要做到的，不容迟疑。**

而在不同工具中，设置连续递增的方式是不同的。

LoadRunner 设计如下：



JMeter 设计如下：

总之，请你记住，在设计场景中，我们一定要做到上面这种连续递增的样子。

业务场景

在 RESAR 性能工程中，性能场景只需要这四类即可：

执行顺序先后为：基准场景、容量场景、稳定性场景、异常场景。

请你注意，除了这四类性能场景外，再没有其他类型的场景了。在每一个场景分类中，我们都可以设计多个具体的场景来覆盖完整的业务。

下面我给你一一解释一下。

1. 基准场景

我经常看到有人说，用脚本加上三五个线程跑上多少次迭代，就算是基准场景了。你可以想想这样的场景意义何在？它仅能验证一下脚本和场景是正确的而已。所以，我不把这样的步骤称为基准场景。

在我的 RESAR 性能工程理念中，**基准场景必须是容量场景的前奏**。具体怎么做呢？那就是在基准场景中，我们也要通过递增连续的场景做到最大 TPS。也就是说在基准场景中，我们要把单接口或单业务压到最大 TPS，然后来分析单接口或单业务的瓶颈点在哪里。

可能你会问，在基准场景中有没有必要做调优的动作呢？

根据我的经验，应该先判断当前单接口或单业务的最大 TPS，有没有超过目标 TPS。如果超过，并且响应时间也在业务可接受的范围之内，那就不用调优。如果没有超过，那必须要做调优。

另外，根据 RESAR 性能工程理论，性能执行的第一阶段目标就是把资源用光，第二阶段的目标是将系统优化到满足业务容量。要知道，任何一个系统要调优都是无止境的，而我们的目标是要保证系统的正常运行。

因为在我们这个课程的示例项目中只有一个系统，所以，我们先做接口级的，然后把接口拼装成完整的业务量，并实现业务模型，然后再在容量场景中执行。在这里，我们将执行测试范围中接口的基准场景。

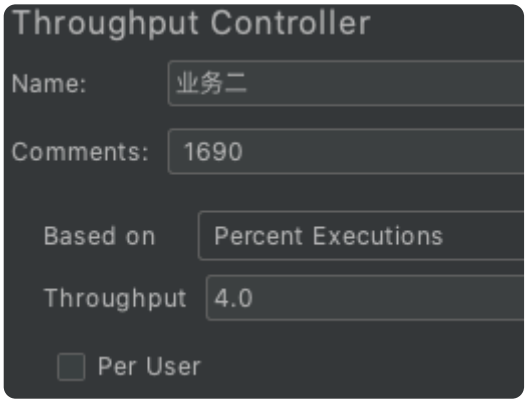
2. 容量场景

有了基准场景的结果之后，我们就进入了容量场景的阶段。在容量场景中，我们还是要继续秉承“连续、递增”的执行思路，最重要的是，要实现我们前面提到的业务模型，来真实模拟线上的业务场景。

我们可以经常看到，现在很多的性能项目里，大部分性能需求都提得不是很具体，从而导致性能场景的模型和生产场景不一致，这是一个严重的问题。

还有一个严重的问题是，即便业务模型和生产一致了，也会由于性能工程师在执行过程中没有严格模拟业务模型中的比例，性能场景的结果变得毫无意义。要知道，在执行过程中，响应时间会随着压力的增加而增加，我们仅用线程数来控制比例是非常不理智的，因为在执行的过程中会出现业务比例失衡。

那应该如何控制这个比例关系呢？如果你是用 JMeter 的话，可以使用 Throughput Controller 来控制业务比例，如下所示：



当然，你也可以用其他方式来实现。总之，在场景执行结束之后，我们要把业务比例做统计，并且要和业务比例对比，当比例一致时，才算是合理的场景。

在容量场景中，我们还有一个要确定的事情，就是什么是最大的 TPS。

我想请你看一下这张图，你觉得最大的 TPS 是多少呢?

你是不是想说最大的 TPS 是 700?

不管你给出的是不是这个答案，在我的性能理念中，我都想跟你强调一点：**容量场景的最大 TPS 是指最大的稳定 TPS。**

那么你看，上面这张图已经抖动了，已经不稳定了，我们再去找它的最大 TPS 还有什么意义呢？你敢让一个生产系统运行在这样抖动的状态中吗？所以，对于上图中这样的 TPS 曲线，我会把最大的稳定 TPS 定为第三个阶梯，也就是在 600 左右，而不会定在 700。

另外，也请你注意，在性能场景中，特别是在容量场景中，经常有人提到“性能拐点”这个词，并且把性能拐点称作是判断性能瓶颈的关键知识。对此，我先不做评判，我们来看一下什么是拐点，它在数学中的定义是这样的：

> 拐点，又称反曲点，在数学上指改变曲线向上或向下方向的点。直观地说，拐点是使切线穿越曲线的点（即连续曲线的凹弧与凸弧的分界点）。

那么在 TPS 曲线中，你真的能找到这样的点吗？反正我是找不到。就以我们上面那张图为例，图中哪里是拐点呢？也许有人会说这个曲线没有拐点。咳咳，那就没得聊了……

可见，性能拐点其实是一个在具体执行过程中非常有误导性的概念。请你以后尽量不要再用“性能拐点”这个词来尝试描述性能的曲线，除非你是真的看到了拐点。

3. 稳定性场景

在完成了容量场景之后，我们就要进入稳定性场景的阶段了。到现在为止，在性能的市场中，还没有人能给出一个稳定性场景应该运行多长时间的确切结论。我们知道根据业务属性不同，稳定性场景也有不同的设计思路，可是这样说起来未免有些空泛。所以，我在这里给出一个稳定性场景的运行指导思路。

在稳定性场景中，我们只有两个关键点：

第一个关键点：稳定性场景的时长。

关于稳定性场景的时间，我经常看到网上有人说一般运行两小时、7*24 小时之类的话。可是，什么叫“一般”，什么又叫“不一般”呢？作为从业十几年的老鸟，我从来没有按照这样的逻辑执行过，也从来没有看到过这些运行时长的具体来源，只看到过很多以讹传讹的文章。

在性能领域中，这样的例子实在太多了，现在我也见怪不怪，毕竟保持本心做正确事情最为重要。下面我给你解释一下什么才是合理的稳定性场景时长。

我们知道，一个系统上线之后，运维人员肯定会做运维巡检，如果发现有问题就会去处理。有的系统是有固定的运维周期，比如说会设定固定的 Job 来做归档之类的动作；有的

系统是根据巡检的结果做相应的动作。

而稳定性要做的就是保证在运维周期之内业务可以正常。 所以，在性能的稳定性场景中，我们要完全覆盖业务容量。比如说对于下面这张图：

在运维周期内，有 1 亿笔业务容量。根据上面容量场景中的测试结果，假设最大稳定 TPS 是 500，那稳定性场景的执行时长就是：

$$稳定性时长 = 100000000 \div 500 \div 3600 \div 24 \approx 2.3(天)$$

通过这样的计算，我们就能知道稳定性场景应该跑多长时间，这也是唯一合理的方式。

第二个关键点：用多大的 TPS 来执行。

对此，我看到网上有人提到，用最大 TPS 的 80% 来运行稳定性场景。这里我不禁要问了，为什么？凭什么不能用最大的来运行呢？

记得我在做培训的时候，有过多次这样的讨论。有人说，之所以用最大 TPS 的 80%，是因为在执行稳定性场景时不能给系统太大的压力，否则容易导致系统出现问题。

这种说法就奇怪了。既然容量场景都能得出最大的 TPS，为什么稳定性就不能用呢？如果用最大的 TPS 执行稳定性场景会出现问题，那这些问题不正是我们希望测试出来的性能问题吗？为什么要用低 TPS 来避免性能问题的出现呢？

所以，用最大 TPS 的 80% 来做稳定性场景是一个错误的思路。

在我的性能理念中，**在执行稳定性场景时，完全可以用最大的稳定 TPS 来运行，只要覆盖了运维周期之内的业务容量即可**。如果你不用最大的稳定 TPS 来运行，而是用低 TPS 来运行，那也必须要覆盖运维周期之内的业务容量。

讲到这里，我觉得上述内容足以指导你做出正确合理的稳定性场景测试了。

4. 异常场景

对于异常场景，有些企业是把它放到非功能场景分类中的，这个我倒觉得无所谓。不管放在哪里都是要有人执行的。我之所以把异常场景放在性能部分，是因为这些异常场景需要在有压力的情况下执行。

对于常规的异常场景，我们经常做的就是：

宕主机；

宕网卡；

宕应用。

除此之外，在现在微服务盛行的时代，我们还有了新的招——宕容器。

当然，你也可以用一些所谓的“混沌工程“的工具来实现对容器的随机删除、网络丢包、模拟 CPU 高等操作，不过，这就是一个大话题了。在这后面的课程里，我会设计几个常用的异常性能场景来带你看下效果。

监控设计

全局监控

其实，有了前面的监控工具部分，监控设计就已经出现写方案之人的脑子里了。对于我们这个课程所用的系统，全局监控如下所示：

从上图来看，我们使用 Prometheus/Grafana/Spring Boot Admin/SkyWalking/Weave Scope/ELK/EFK 就可以实现具有全局视角的第一层监控。对工具中没有覆盖的第一层计数器，我们只能在执行场景时再执行命令来补充了。

定向监控

那后面的定向监控怎么办呢？在这里我也大体列一下常用的工具。不过，请你注意，这些工具是在有问题的时候才会去使用。

其实在性能分析中，除了表格中的这三个工具之外，还有很多工具会在查找性能瓶颈证据链时使用，我在这里无法全部罗列出来，只能根据系统使用到的技术组件，大概罗列一下我能想到的常用工具。在后续课程的操作中，如果你发现我们用了表格中没有列出的工具，也请你不要惊讶。

项目组织架构

在性能方案中，我们一定要画出项目的组织架构图，并且请你一定要在这部分写明各组织人员的工作范围和职责，避免出现扯皮的情况。我大体画一下常见的组织架构：

这是我按照事情来划分的，而非职场中的工作职位，我觉得这是一个合理的组织架构。在这张图中，性能脚本工程师所负责的事情，其实是现在大部分性能从业人员都在做，并且仅仅在做的事。至于性能分析工程师，在很多性能项目中几乎不存在，也没有这样的固定职位。其实，性能分析工程师很有必要存在。

此外，架构师、开发工程师、运维工程师都需要在支持性能分析的状态。请注意，我说的“支持”，并不是指站在旁边看着，而是在有了问题之后，要能具体地给出支持，而非推诿责任。

业务方是性能的业务需求来源，这是一定要有的。如果业务方提不出来什么合理的性能需求，那这个项目基本上会是稀碎的样子。

至于老板这个角色，在性能项目中，我经常看到的老板都不懂什么叫性能，只会叫着要支持 XXX 并发用户数，支持 XXX 在线用户数。其实，这样的老板沟通起来也很简单，就是拿结果给他就好了。不过，在性能项目的执行过程中，当资源不足时，请你一定要让老板知道，同时降低老板的预期，要不然在后续的沟通中会非常费劲。

成果输出

过程性输出

1. 脚本
2. 场景执行结果
3. 监控结果

4. 问题记录

在性能项目中，过程性输出有这些内容就够了，不用更多，当然，也不能更少了。我经常看到很多性能项目在执行完之后，除了有一份性能测试报告之外，什么过程性输出都没有。我实在不理解这样的企业是怎么积累性能经验的。所以，我还是要规劝你一句，**在性能项目中，尽量多做一些归档整理的工作，以备在后面的项目中查阅，并实现自己的技术积累。**

结果输出

通常情况下，在我做的性能项目中，都会输出两个最终交付的文档：一个是性能场景执行结果记录的报告（就是现在我们常写的性能测试报告），另一个是性能调优报告。

性能项目测试报告

性能项目测试报告想必大家见得多了，这里我只强调几点：

1. 性能结果报告一定要有结论，而不是给出一堆“资源使用率是多少”、“TPS 是多少”、“响应时间是多少”这种描述类的总结语。你想想，性能结果都在这个报告中了，谁还看不见怎么滴？还要你复述一遍吗？我们要给出“当前系统可支持 XXX 并发用户数，XXX 在线用户数”这样的结论。
2. 一定不要用“可能”、“或许”、“理应”这种模棱两可的词，否则就是在赤裸裸地耍流氓。
3. 性能结果报告中要有对运维工作的建议，也就是要给出关键性能参数的配置建议，比如线程池、队列、超时等。
4. 性能结果报告中要有对后续性能工作的建议。

性能调优报告

为什么我要强调单独写调优报告呢？因为**调优报告才是整个性能项目的精华，调优报告中一定要记录下每一个性能问题的问题现象、分析过程、解决方案和解决效果**。可以说，调优报告完全是一个团队技术能力的体现。

项目风险分析

对于性能项目的风险，我把比较常见的风险列在这里：

1. 业务层的性能需求不明确
2. 环境问题
3. 数据问题
4. 业务模型不准确
5. 团队间协调沟通困难
6. 瓶颈分析不到位，影响进度
7. ……

在我们这个课程所用的项目中，比较大的风险就是：

1. 硬件资源有限。
2. 项目时间不可控，因为出了问题，并没有人支持，只能自己搞。

不过请你放心，我会努力克服困难，把这个项目的执行过程都记录下来。

到这里，整个性能项目实施方案就结束了。如果你认真看到了这里，那么恭喜你，你已经超越了很多人。我为你点赞！

总结

在这节课里，我把一个性能方案该有的内容以及要写到什么程度，都给你梳理了一遍。希望能给你一些借鉴。

性能方案是一个性能项目的重要输出。如果你是在项目中做快速迭代，可能并不需要写如此复杂并且重量级的文档。因为文档里描述的很多工作都已经做过了，你可能只需要跟着版本去做迭代比对就好了。

但对于一个完整的项目来说，性能方案就显得极为重要。因为它指导了这个项目的整个过程。在性能方案中，我们强调了几个重点：业务模型、性能指标、系统架构图、场景设计、监控设计等，它们都会对整个项目的质量起到关键作用。

最后，我希望你可以在后续的项目中，尝试去写一个完整的性能方案。

课后作业

学完这节课，请你思考两个问题：

1. 如何精确模拟业务模型？
2. 为什么我们要强调系统架构图的重要性？

欢迎你在留言区与我交流讨论。当然了，你也可以把这节课分享给你身边的朋友，他们的一些想法或许会让你有更大的收获。我们下节课见！

关于课程读者群

点击课程详情页的链接，扫描二维码，就可以加入我们这个课程的读者群哦，希望这里的交流与思维碰撞能帮助你取得更大的进步，期待你的到来～

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04 | 如何构建性能分析决策树和查找瓶颈证据链？

下一篇 06 | 如何抽取出符合真实业务场景的业务模型？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。